Adventures In Real-Time Performance Tuning, Part 2

The real-time for Linux patchset does not guarantee adequate real-time behavior for all target platforms. When using real-time Linux on a new platform you should expect to have to tune the kernel and drivers to provide performance that matches your specific requirements.

Part 1, presented at ELC 2008, provided an example of the trials and tribulations of the tuning journey for a MIPS target board.

Part 2 will provide additional examples of methods to debug and tune latency. An additional target for this installment is an SMP ARM board, leading to a new set of challenges.

Frank Rowand, Sony Corporation of America November 7, 2008

Adventures In Real-Time Performance Tuning, Part 2

With thanks to the Sony SS kernel team in Tokyo for their contributions to this tuning and debugging process.

Overview

Background and Definitions

Some Tuning Strategies and Tactics

Examples of Tuning

Quick Summary of Part 1

Examples of using the old latency instrumentation from the RT patchset to tune latency

/proc/latency_hist/interrupt_off_latency/CPU*

/proc/latency_trace

The new, improved “ftrace” instrumentation appears in the RT patchset 2.6.24-rt2 and mainline 2.6.27-rc1

Quick Summary of Part 1

Characterization of the (large) overhead of latency instrumentation

Tuning a component of overall latency (micro-tuning) can have a negative impact on the overall latency

The value of data visualization (graphs) vs. basic statistics such as min, max, avg, std deviation.

What Is Different

part 1

Linux 2.6.24, MIPS target, UP

part 2

Linux 2.6.23, ARM target, SMP

What is Real Time?

It is determinism (being able to respond to a stimulus before a deadline) within a given system load envelope.

It is NOT fast response time.

The specific real time application deadlines determine how short the maximum response time must be to deliver real time behavior.

Some examples of deadlines are one second, one millisecond, or five microseconds.

RT latency is the delay from stimulus to when the RT “application” is executing code

RT latency is the delay from stimulus to when the RT “application” is executing code

Possible RT application contexts include

- driver interrupt context
- driver thread context
- kernel thread context
- user space thread context

RT latency is the delay from stimulus to when the RT “application” is executing code

The components that add up to RT latency are important to the tuning process, but keep in mind the goal of tuning actual RT latency.

Some components that may contribute to RT latency

- IRQ disabled time
- preempt disabled time
- IRQ latency, from event until bottom half
- RT driver bottom half(s) execution
- non-RT driver bottom half(s) execution
- task switch time

Some components that may contribute to RT latency

- IRQ disabled time
- preempt disabled time
- IRQ latency, from event until bottom half
- RT driver bottom half(s) execution
- non-RT driver bottom half(s) execution
- task switch time

The components that add up to RT latency are important to the tuning process, but keep in mind the end goal of tuning actual RT latency.

Strategy: Compare Kernel Versions

Use Case #1

I'm stuck on kernel version 2.6.n, but there are reports on the linux-rt-users email list that performance on version 2.6.n + 3 is much better.

Is there some fix that I can port to my kernel version?

Strategy: Compare Kernel Versions

Use Case #2

I moved forward from kernel version 2.6.n to 2.6.n+1 and performance got worse.

Tactics

#1 Compare kernel config options

Tactics

#1 Compare kernel config options

#2 Compare the behavior and/or performance metrics

Raw Interrupt Latency

Latency of timer interrupt to interrupt bottom half execution.

Raw Interrupt Latency

Latency of timer interrupt to interrupt bottom half execution.

Problem: latency increased from 6 usec to 11 usec
on move from 2.6.22 to 2.6.23

Raw Interrupt Latency

Latency of timer interrupt to interrupt bottom half execution.

Problem: latency increased from 6 usec to 11 usec on move from 2.6.22 to 2.6.23

Disclaimer: all kernel versions in this presentation are based on kernel.org, but with patches added, so these results may not be repeatable on kernel.org versions

Tactics

#1 Compare kernel config options

#2 Compare the behavior and/or performance metrics

Revert the config differences, starting with the most likely culprits.

Metric: raw interrupt latency.

Results

option	value	usec
NR_CPUS	4 -> 2	9/10 -> 9
NO_HZ	y -> n	9 -> 6
PREEMPT_RCU_BOOST	y -> n	6 -> 5
SLAB -> SLOB		5/6 -> 5
others		no big affect

Task Wake Up Time

Goal: average task wake up time < XXX usec

Task Wake Up Time

Goal: average task wake up time < XXX usec

Problem: actual task wake up time >> XXX usec

Task Wake Up Time

Goal: average task wake up time < XXX usec

Problem: actual task wake up time >> XXX usec

Observation:

```
 2.6.22   avg: 23 usec   max:  33 usec
 2.6.23   avg: 72 usec   max: 244 usec
```

Tactics

#1 Compare kernel config options

#2 Compare the behavior and/or performance metrics

#3 Compare the source code
- the real-time patch set
- the base kernel

#3b Read the current version of the source code

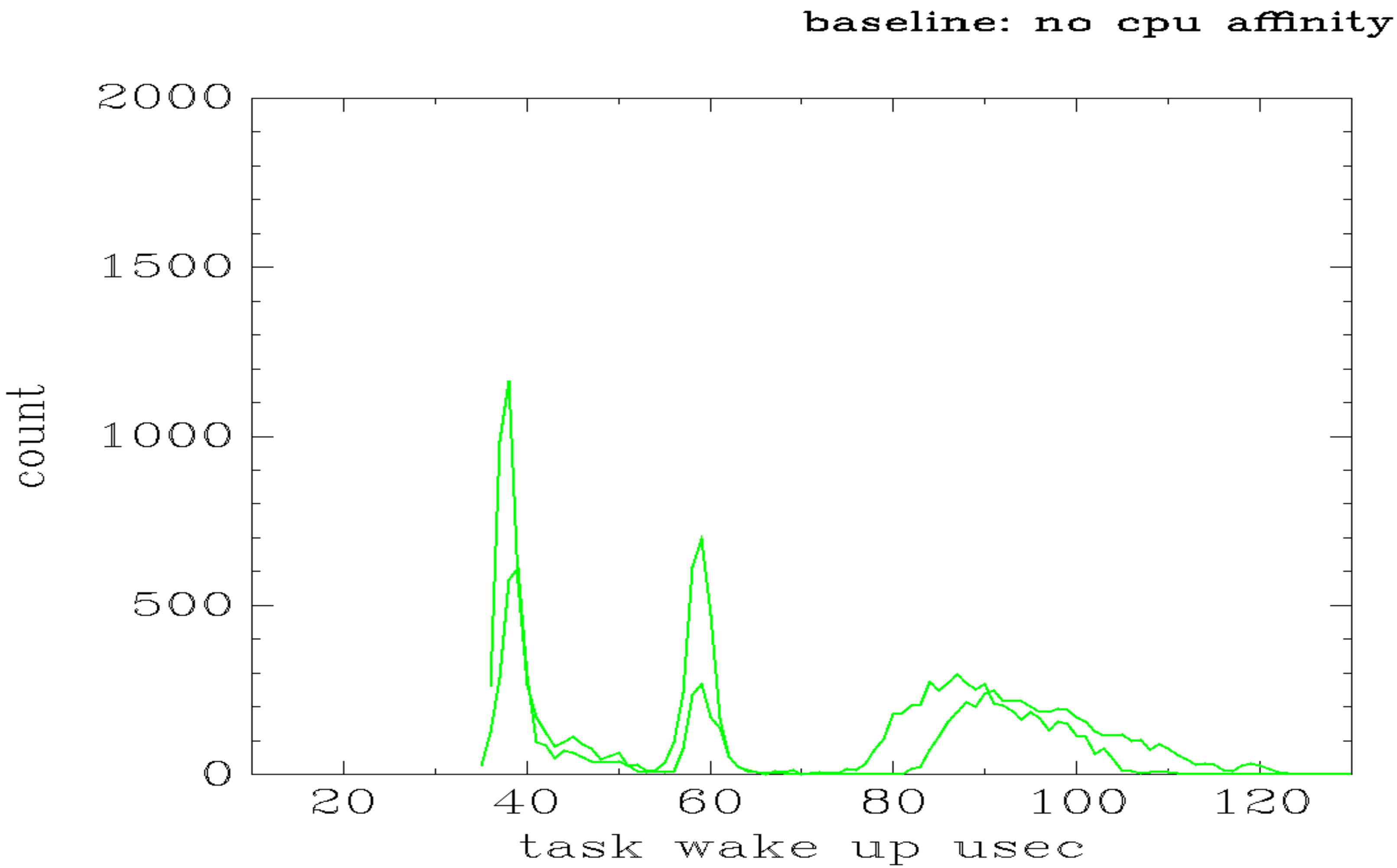
Task Wake Up Test (simplified)

```
/* producer kernel thread, SCHED_FIFO, priority = 98 */

pthread() {
   counter = 10000;

   while (counter-- > 0) {
      mbx->timestamp = read_time();
      set_current_state(TASK_UNINTERRUPTIBLE);
      wake_up_process(ctsk);
      schedule_timeout(DELAY);
   }

   done = 1;
   wake_up_process(ctsk);
}
```

Task Wake Up Test (simplified)

```
/* consumer kernel thread, SCHED_FIFO, priority = 99 */

cthread() {

   while (1) {
      delta = read_time() - mbx->timestamp;
      update_stats(delta);
      set_current_state(TASK_UNINTERRUPTIBLE);
      if (done)
         break;
      schedule();
   }

   report_stats();
}
```

2.6.23 Task Wake Up Time



Intuitive Leap 1

Which processor is each thread on?

Intuitive Leap 1

Which processor is each thread on?

(Ignoring the strategy and tactics for a moment!)

Methodology (M1)

(M1) Instrument the test application

- Add mbx->cpu
  - Producer sets to current cpu
  - Consumer compares to current cpu

Result (M1)

```
                 ---   producer  ---
consumer          cpu 0         cpu 1
--------         ----------  ----------
 cpu 0                    0        5000
 cpu 1                 4999           1
```

This may be good - each task is on its own processor.

Result (M1)

```
                 ---   producer   ---
consumer          cpu 0          cpu 1
--------         -----------  -----------
 cpu 0                     0         5000
 cpu 1                  4999            1
```

This may be good - each task is on its own processor.

Next Question:
How frequently did producer move between cpus?

Methodology (M2)

(M2) Instrument the test application some more

Log producer and consumer cpu for each iteration

Result (M2)

```
# producer cpu map:
#
#       0 0101010101010101010101010101010101010101010101010101010101010101010101
#      70 0101010101010101010101010101010101010101010101010101010101010101010101
#     140 0101010101010101010101010101010101010101010101010101010101010101010101
#     210 0101010101010101010101010101010101010101010101010101010101010101010101
#     280 0101010101010101010101010101010101010101010101010101010101010101010101
#     350 0101010101010101010101010101010101010101010101010101010101010101010101
#     420 0101010101010101010101010101010101010101010101010101010101010101010101
#     490 0101010101010101010101010101010101010101010101010101010101010101010101
#     560 0101010101010101010101010101010101010101010101010101010101010101010101
#     630 0101010101010101010101010101010101010101010101010101010101010101010101
#     700 0101010101010101010101010101010101010101010101010101010101010101010101
#
# consumer cpu map:
#
#       0 1010101010101010101010101010101010101010101010101010101010101010101010
#      70 1010101010101010101010101010101010101010101010101010101010101010101010
#     140 1010101010101010101010101010101010101010101010101010101010101010101010
#     210 1010101010101010101010101010101010101010101010101010101010101010101010
#     280 1010101010101010101010101010101010101010101010101010101010101010101010
#     350 1010101010101010101010101010101010101010101010101010101010101010101010
#     420 1010101010101010101010101010101010101010101010101010101010101010101010
#     490 1010101010101010101010101010101010101010101010101010101010101010101010
#     560 1010101010101010101010101010101010101010101010101010101010101010101010
#     630 1010101010101010101010101010101010101010101010101010101010101010101010
#     700 1010101010101010101010101010101010101010101010101010101010101010101010
```

Result (M2)

```
# producer cpu map:
#
#      0 0101010101010101010101010101010101010101
#     70 0101010101010101010101010101010101010101
#
# consumer cpu map:
#
#      0 1010101010101010101010101010101010101010
#     70 1010101010101010101010101010101010101010
```

Result (M2)

```
# producer cpu map:
#
#      0 0101010101010101010101010101010101010101
#     70 0101010101010101010101010101010101010101
#
# consumer cpu map:
#
#      0 1010101010101010101010101010101010101010
#     70 1010101010101010101010101010101010101010
```

Excessive migration is usually not good.

Compare to 2.6.22

```
             ---   producer  ---
consumer      cpu 0       cpu 1
---------    ----------  ----------
 cpu 0           10000           0
```

(Returning to the strategy and tactics.)

Compare to 2.6.22

```
# producer cpu map:
#
#       0 0000000000000000000000000000000000000000
#      70 0000000000000000000000000000000000000000
#
# consumer cpu map:
#
#       0 0000000000000000000000000000000000000000
#      70 0000000000000000000000000000000000000000
```

Fix (F1)

Set processor affinity for pthread() and cthread()

Fix (F1)

Set processor affinity for cthread() and pthread()

Goal: verify that task migration is the cause of the increased task wake up time.

Fix Result (F1)

```
producer   consumer   task switch usec
cpu        cpu        ----------------
affinity   affinity   avg   max
--------   --------   ----  ----
 x          x           69   147   baseline
 x          x           68   159   baseline
```

Fix Result (F1)

```
producer   consumer   task switch usec
cpu        cpu        -----------------
affinity   affinity   avg   max
--------   --------   ----  ----
 x          x           69   147   baseline
 x          x           68   159   baseline
```

Each line is a 10000 iteration test run result.

This data does not match the original problem statement (avg 72, max 244) due to changes in kernel configuration.

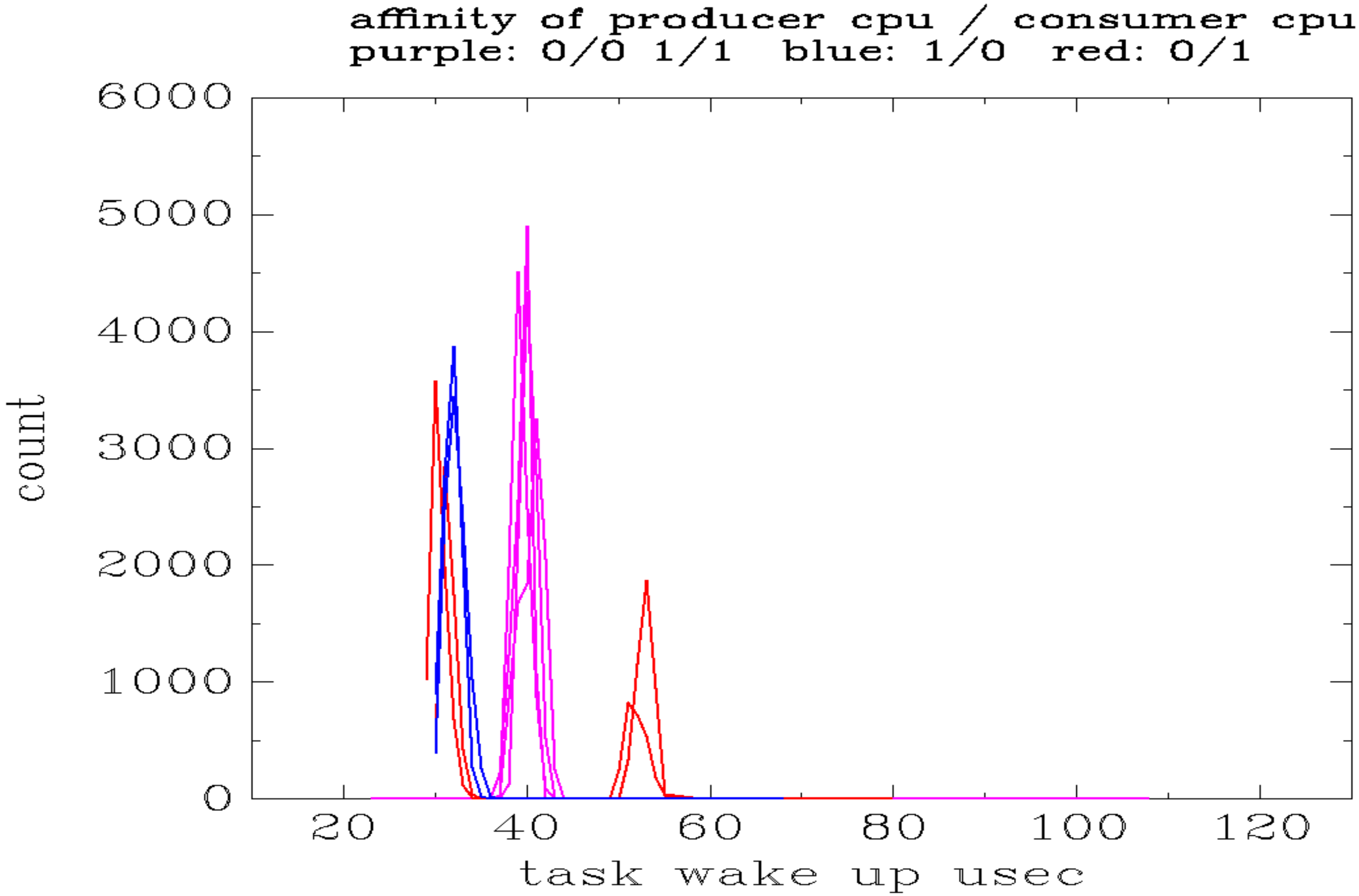
Fix Result (F1)

```
producer   consumer   task switch usec
cpu        cpu        -----------------
affinity   affinity   avg   max
--------   --------   ----  ----
 x          x           69   147   baseline
 x          x           68   159   baseline

 0          0           39    64
 0          0           40   108
```

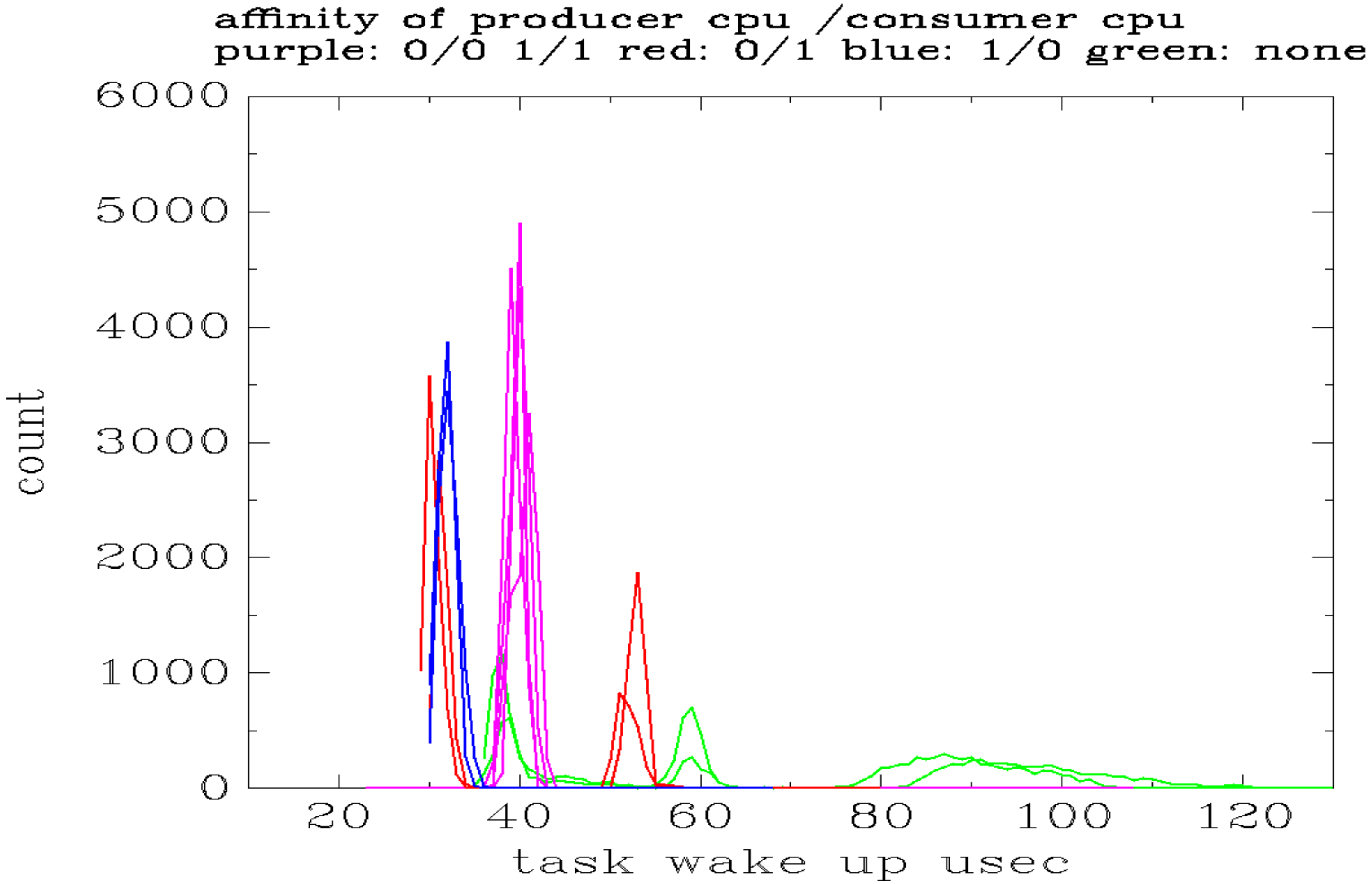
Fix Result (F1)

```
producer   consumer   task switch usec
cpu        cpu        ----------------
affinity   affinity   avg   max
--------   --------   ----  ----
 x          x           69   147   baseline
 x          x           68   159   baseline

 0          0           39    64
 0          0           40   108

 1          1           40    94
 1          1           39    45

 0          1           40    66
 0          1           36    80

 1          0           31    68
 1          0           32    68
```

Fix Result (F1)



Fix Result (F1)



Fix Conclusions (F1)

Processor migration increases task wake up time significantly.

Fix Conclusions (F1)

Processor migration increases task wake up time significantly.

But setting processor affinity for a large number of real-time tasks is not a desired solution for the problem.

Fix Conclusions (F1)

Processor migration increases task wake up time significantly.

But setting processor affinity for a large number of real-time tasks is not a desired solution for the problem.

And task wake up time still larger than on 2.6.22.

Recall Tactic #1

Examine how the kernel source changed.

Recall Tactic #1

Examine how the kernel source changed.

The real-time scheduler became more aggressive about moving tasks between processors.

Recall Tactic #1

Examine how the kernel source changed.

The real-time scheduler became more aggressive about moving tasks between processors.

Possible fix: revert to the old scheduler algorithm.

Recall Tactic #1

Examine how the kernel source changed.

The real-time scheduler became more aggressive about moving tasks between processors.

Possible fix: revert to the old scheduler algorithm.

But moving to older versions is not the direction that I want to go...

So attempt to improve the current version.

2.6.23 Real Time Scheduler

When a real time task wakes a higher priority real time task the real-time scheduler prefers to not push the (lower priority) currently running task to another processor.

2.6.23 Real Time Scheduler

When a real time task wakes a higher priority real time task the real-time scheduler prefers to not push the (lower priority) currently running task to another processor.

The assumption is that the running process is cache hot and the newly awakened process is cache cold.

Revisit Test

```
P:  producer thread, priority = 98
C:  consumer thread, priority = 99
TI: timer interrupt handler


P [98]: set_current_state(TASK_UNINTERRUPTIBLE)
P [98]: wake_up_process(consumer)
   C: [99]: consumer may preempt producer
            or may be pushed to other processor
   C: [99]: processes message
   C: [99]: schedule()
P [98]: schedule_timeout()
      TI [--]: timer irq, wake_up_process(producer)
P [98]: create next message
```

Fix (F2)

May allow consumer to remain on the same processor as the producer.

```
select_task_rq_rt()

+        /*
+         * If current task on this CPU is about to sleep,
+         * next task should run on this CPU.
+         */
+        if (current->state != TASK_RUNNING) {
+                int cpu = smp_processor_id();
+                if (cpu_isset(cpu, p->cpus_allowed))
+                        return cpu;
+        }
```

Fix Result (F2)

producer cpu	consumer cpu	task wake up usec avg	task wake up usec max	
x	x	69	147	baseline
x	x	68	159	baseline
0	0	39	64	Fix F1
0	0	40	108	Fix F1
1	1	40	94	Fix F1
1	1	39	45	Fix F1
x	x	23	63	Fix F2
x	x	23	65	Fix F2

Fix Result (F2)

producer and consumer always on same cpu,
instead of always on the other cpu

```
#                 -- producer ------------
# consumer          cpu 0         cpu 1
# --------        ----------   ----------
#  cpu 0                5168            0
#  cpu 1                   0         4832
```

Fix Result (F2)

producer and consumer always on same cpu,
instead of always on the other cpu

```
#                 -- producer -------------
# consumer          cpu 0          cpu 1
# --------        ----------    ----------
#  cpu 0                5168             0
#  cpu 1                   0          4832
```

How often does migration occur?
  One migration per test run?
  One migration per message?

Result (M2)

```
# producer cpu map:
#
#       0 0010101010000000000100111111111111111111111111111111111111111111111111
#      70 1111111111111111111111111111111111111111111111111100000000000000000000
#     140 0000000000001111111111111111111111111111111111111111111111111111111111
#     210 1111111110001111111111111111111111111111111111111111111111111111111111
#     280 1111111111111111111111111111111111111111000000000000000000000000000000
#     350 0000000000000000000000000000000000000000000000000000000000000000000001
#     420 0011111111111111111111111111111111111111111111111111111111111111111111
#     490 1111111111111111111111111111110000000000000000000000000000000000000000
#     560 0000000000000000000000000000000000000000000000000000000000010011111111
#     630 1111111111111111111111111111111111111111111111111111111111111111111111
#     700 1111111111111111111100000000000000000000000000000000000000000000000000
#
# consumer cpu map:
#
#       0 0010101010000000000100111111111111111111111111111111111111111111111111
#      70 1111111111111111111111111111111111111111111111111100000000000000000000
#     140 0000000000001111111111111111111111111111111111111111111111111111111111
#     210 1111111110001111111111111111111111111111111111111111111111111111111111
#     280 1111111111111111111111111111111111111111000000000000000000000000000000
#     350 0000000000000000000000000000000000000000000000000000000000000000000001
#     420 0011111111111111111111111111111111111111111111111111111111111111111111
#     490 1111111111111111111111111111110000000000000000000000000000000000000000
#     560 0000000000000000000000000000000000000000000000000000000000010011111111
#     630 1111111111111111111111111111111111111111111111111111111111111111111111
#     700 1111111111111111111100000000000000000000000000000000000000000000000000
```

Result (M2)

```
# producer cpu map:
#
#        0 00101010100000000001001111111111111111
#       70 11111111111111111111111111111111111111
#      140 00000000000011111111111111111111111111
#
# consumer cpu map:
#
#        0 00101010100000000001001111111111111111
#       70 11111111111111111111111111111111111111
#      140 00000000000011111111111111111111111111
```

Occasional migration, but producer and consumer always on same cpu

Revisit Test, in more detail

```
P:  producer thread, priority = 98
C:  consumer thread, priority = 99
TI: timer interrupt handler, irq context
TT: timer interrupt handler, thread context, priority = 50

P [98]: set_current_state(TASK_UNINTERRUPTIBLE)
P [98]: wake_up_process(consumer)
   C: [99]: consumer preempts producer
            producer may be pulled by other processor
   C: [99]: processes message
   C: [99]: schedule()
P [98]: schedule_timeout()
      TI [--]: timer irq
         TT [50]: wake_up_process(producer)
                  producer may be pushed to other processor
P [98]: create next message
```

Revisit Test, in more detail

Possible double migration for producer.

```
P:  producer thread, priority = 98
C:  consumer thread, priority = 99
TI: timer interrupt handler, irq context
TT: timer interrupt handler, thread context, priority = 50

P [98]: set_current_state(TASK_UNINTERRUPTIBLE)
P [98]: wake_up_process(consumer)
   C: [99]: consumer preempts producer
            producer may be pulled by other processor
   C: [99]: processes message
   C: [99]: schedule()
P [98]: schedule_timeout()
      TI [--]: timer irq
         TT [50]: wake_up_process(producer)
                  producer may be pushed to other processor
P [98]: create next message
```

Methodology (M3)

Verify double migration theory

Result (M3)

Adding further instrumentation to the producer task verified the suspected additional task migration.

Fix (F3)

Modify kernel/sched_rt.c algorithms to
make real-time process migration less aggressive

Fix (F3)

Define “overloaded”, for purposes of pushing RT tasks, as:

number of tasks on RT run queue

>= CONFIG_RT_OVERLOAD

Fix (F3)

Define “overloaded”, for purposes of pushing RT tasks, as:

number of tasks on RT run queue
with priority > (MAX_USER_RT_PRIO / 2)

>= CONFIG_RT_OVERLOAD

Fix (F3)

Define “overloaded”, for purposes of pushing RT tasks, as:

number of tasks on RT run queue
with priority > (MAX_USER_RT_PRIO / 2)

>= CONFIG_RT_OVERLOAD

instead of

number of tasks on RT run queue > 0

Fix (F3)

Define “overloaded”, for purposes of pushing RT tasks, as:

number of tasks on RT run queue
with priority > (MAX_USER_RT_PRIO / 2)

>= CONFIG_RT_OVERLOAD

The priority of many kernel threads is:

MAX_USER_RT_PRIO / 2

```
#define MAX_USER_RT_PRIO CONFIG_MAX_USER_RT_PRIO

MAX_USER_RT_PRIO / 2  ==  50

  PID CMD                                  RTPRIO CLS

    3 [migration/0]                            99 FF
    4 [posix_cpu_timer]                        99 FF
    5 [softirq-high/0]                         50 FF
    6 [softirq-timer/0]                        50 FF
    7 [softirq-net-tx/]                        50 FF
    8 [softirq-net-rx/]                        50 FF
    9 [softirq-block/0]                        50 FF
   10 [softirq-tasklet]                        50 FF
   11 [softirq-sched/0]                        50 FF
   12 [softirq-rcu/0]                          50 FF
   13 [watchdog/0]                             99 FF
   27 [events/0]                                1 FF
   71 [krcupreemptd]                            1 FF
   94 [IRQ-125]                                50 FF
  103 [IRQ-152]                                99 FF
```

Fix Result (F3)

```
                                 task wake up usec
producer        consumer         -----------------
cpu             cpu              avg      max
--------        --------         ----     ----
 x               x                 69      147    baseline
 x               x                 68      159    baseline

 x               x                 23       63    Fix F2
 x               x                 23       65    Fix F2

 x               x                 23       33    2.6.22

 x               x                 21       41    Fix F3
 x               x                 21       25    Fix F3
```

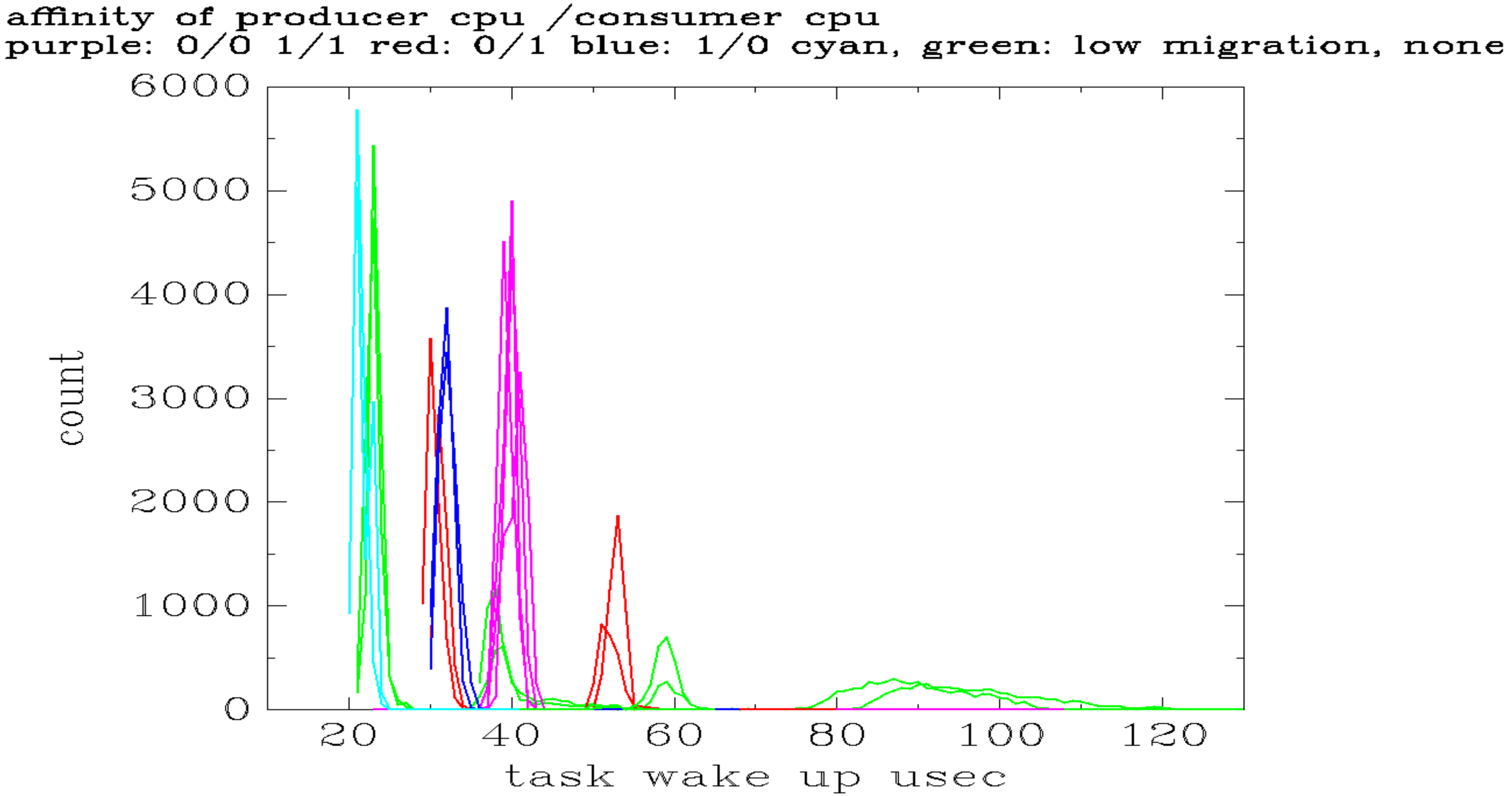
Fix Result (F3)

producer cpu	consumer cpu	task wake up usec avg	task wake up usec max	
x	x	69	147	baseline
x	x	68	159	baseline
x	x	23	63	Fix F2
x	x	23	65	Fix F2
x	x	23	33	2.6.22
x	x	21	41	Fix F3
x	x	21	25	Fix F3

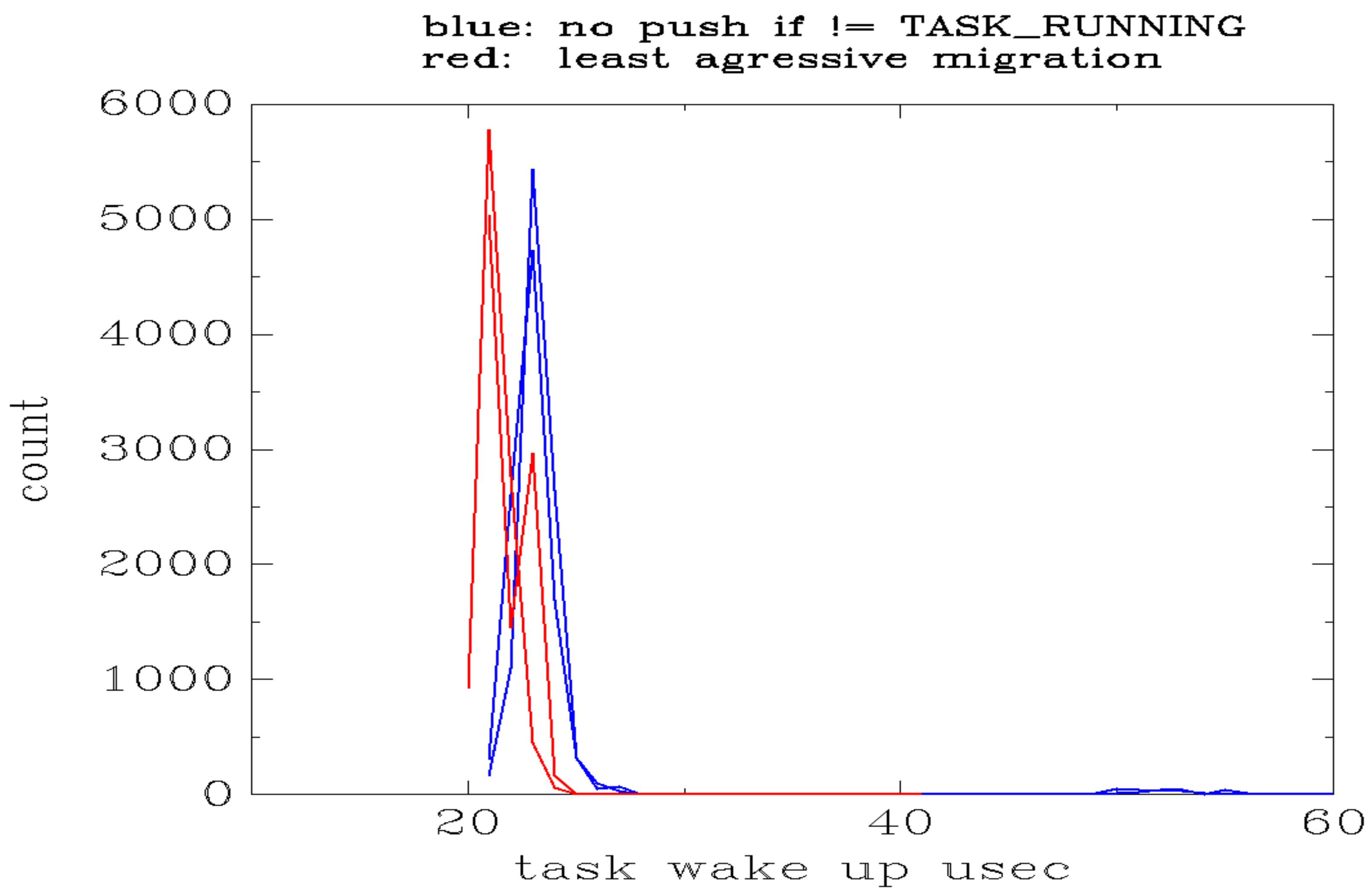
additional checks:

- average irq/preempt off improved
- maximum irq/preempt off improved
- cyclictest results not negatively impacted

Fix Result (F2, F3)



Fix Result (F2, F3)



Fix Conclusion (F2, F3)

Reducing processor migration may decrease task wake up time significantly.

Fix Conclusion (F2, F3)

Reducing processor migration may decrease task wake up time significantly.

But test is a rather simplistic case

- real-time task processing time << task switch time
- real-time task processing time << task migration time

Fix Conclusion (F2, F3)

Reducing processor migration may decrease task wake up time significantly.

But test is a rather simplistic case

real-time task processing time << task switch time
real-time task processing time << task migration time

Will these improvements apply to a more complex real-time application?

Strategy: Trust Nothing, Blame Everything

Strategy: Trust Nothing, Blame Everything

A minor distraction from both the debug / fix process and the flow of this presentation. But this is about the point in the debug process that I got around to tracking an irritant in the data.

Strategy: Trust Nothing, Blame Everything

A minor distraction from both the debug / fix process and the flow of this presentation. But this is about the point in the debug process that I got around to tracking an irritant in the data.

Real life is never the clean, straight line process that this talk pretends to show.

Data from step n + X is often hard to compare to data from step n.

Strategy: Trust Nothing, Blame Everything

A single very large maximum task wake up time was occasionally appearing during the early stages of the testing and improvements described in the previous slides.

The data for the tables of results and graphs shown on previous slides was recreated after resolving the cause of the large maximum.

Strategy: Trust Nothing, Blame Everything

A single very large maximum task wake up time was occasionally appearing during the early stages of the testing and improvements described in the previous slides.

CAUSE: a printk() in the consumer task, located before the test loop

Tactics

#1 Compare kernel config options

Still have not applied tactic #1 to the task wake up time issue.

Fix (F4)

1) Kernel configuration changes

```
CONFIG_PREEMPT_RCU_BOOST=n
CONFIG_NO_HZ=n
```

2) Disable ARM option to enable interrupts in context_switch().

CONFIG_DISABLE_WANT_INTERRUPTS_ON_CTXSW=y

(Interrupts are enabled then disabled in context_switch() if __ARCH_WANT_INTERRUPTS_ON_CTXSW is defined. This is similar to, but somewhat different than, the experiment of enabling irqs on the return from interrupts path described in “Adventures in Real-Time Performance Tuning, Part 1”.)

Fix (F4)

1) Kernel configuration changes

```
CONFIG_PREEMPT_RCU_BOOST=n
CONFIG_NO_HZ=n
```

2) Disable ARM option to enable interrupts in context_switch().

CONFIG_DISABLE_WANT_INTERRUPTS_ON_CTXSW

(Interrupts are enabled then disabled in context_switch() if __ARCH_WANT_INTERRUPTS_ON_CTXSW is defined. This is similar to, but somewhat different than, the experiment of enabling irqs on the return from interrupts path described in “Adventures in Real-Time Performance Tuning, Part 1”.)

Goal: verify that the config changes to reduce irq off time do not increase task wake up time

Fix Result (F4)

```
Average task wake up time,
10000 iteration test run 8 times per configuration


21 - 22 usec  CONFIG_DISABLE_WANT_INTERRUPTS_ON_CTXSW=y
              CONFIG_PREEMPT_RCU_BOOST=n
              CONFIG_NO_HZ=n
20 - 22 usec  CONFIG_DISABLE_WANT_INTERRUPTS_ON_CTXSW=y
21 - 24 usec  4 migration fixes, after merging unrelated patches
     22 usec  4 migration fixes, before merging unrelated patches
23 - 24 usec  2 migration fixes
40 - 43 usec  pinned to same processor
32 - 53 usec  pinned to different processor
     73 usec  no processor affinity
```

Not a large impact on average task wakeup time,
but a large improvement in irq/preempt off metric

A Bigger Picture Of Metrics (but still a subset of what I track)

AKA keeping the big picture in view

test	cyclictest						irq off						preempt off						irq/preempt off						
	----maximum---			----average---			----maximum---			----average---			----maximum---			----average---			----maximum---			----average---			
num	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	min	avg	max	
00050	53	110	409	49	85	281	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 3
00051	56	160	401	49	86	272	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 3
00052	147	330	618	90	195	460	3	21	499	3	17	356	3	16	364	3	12	326	6	23	543	6	20	352	fix 3
00053	102	306	793	87	189	479	3	19	488	3	17	329	3	15	395	3	12	296	6	23	443	6	20	357	fix 3
00054	49	164	364	46	90	247	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 3 + _CTXSW=y
00055	53	185	391	47	95	268	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 3 + _CTXSW=y
00056	131	289	591	86	179	429	3	24	374	3	20	297	3	14	325	3	11	269	6	21	539	6	19	345	fix 3 + _CTXSW=y
00057	96	258	687	80	172	463	3	23	320	3	21	305	3	14	295	3	11	240	6	22	336	6	19	290	fix 3 + _CTXSW=y
00058	88	188	335	47	78	241	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 4
00059	49	111	314	45	72	240	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	fix 4
00060	96	229	527	83	157	394	3	20	348	3	18	321	3	14	293	2	11	250	6	21	419	6	18	327	fix 4
00061	115	242	791	84	165	459	3	20	444	3	18	327	3	14	345	1	11	265	6	21	413	6	18	344	fix 4
00062	110	268	707	84	166	443	3	21	383	3	18	334	3	14	357	2	11	284	6	22	361	6	18	327	fix 4

What Does It All Mean?

What Does It All Mean?

Keep real time application performance in sight while tuning individual components

What Does It All Mean?

Keep real time application performance in sight while tuning individual components

Watch a vast range of metrics and behaviors for each change

What Does It All Mean?

Keep real time application performance in sight while tuning individual components

Watch a vast range of metrics and behaviors for each change

Look at a large number of statistics for the metrics (for example, minimum, maximum, average, standard deviation)

What Does It All Mean?

Keep real time application performance in sight while tuning individual components

Watch a vast range of metrics and behaviors for each change

Look at a large number of statistics for the metrics (for example, minimum, maximum, average, standard deviation)

Look at graphic representations of metrics

What Does It All Mean?

What is in the scope of a tuning effort?

What Does It All Mean?

What is in the scope of a tuning effort?

EVERYTHING, just like in any debugging effort!

What Does It All Mean?

What is in the scope of a tuning effort?

EVERYTHING, just like in any debugging effort!

- Instrumentation
- Tests
- Kernel
- Drivers
- Real Time Applications
- Other Applications
- External Influences

What Does It All Mean?

Frank's Law of Performance Tools

What Does It All Mean?

Frank's Law of Performance Tools

The performance metric that you need to answer the current question

- is not available from any existing source or tool

What Does It All Mean?

Frank's Law of Performance Tools

The performance metric that you need to answer the current question

- is not available from any existing source or tool

- or is not presented in a meaningful manner

What Does It All Mean?

Frank's Law of Performance Tools

The performance metric that you need to answer the current question

- is not available from any existing source or tool

- or is not presented in a meaningful manner

You will need to write a new tool or leverage an existing tool.

Resources

Rtiwiki
  http://rt.wiki.kernel.org/index.php/Main_Page

rt-user-list
  http://dir.gmane.org/gmane.linux.rt.user

eLinux.org
  http://elinux.org/Real_Time

cyclictest
  http://git.kernel.org/?p=linux/kernel/git/tglx/rt-tests.git;a=summary

Resources

ftrace
  http://people.redhat.com/srostedt/ftrace-tutorial.odp
  kernel source: Documentation/ftrace.txt

hackbench
  http://devresources.linux-foundation.org/craiger/hackbench/

LatencyTOP
  http://www.latencytop.org

“Stress actions - things that will kill realtime performance”
and information about test programs and testing
  http://elinux.org/Realtime_Testing_Best_Practices

Resources

A realtime preemption overview
http://lwn.net/Articles/146861

What's in the realtime tree
http://lwn.net/Articles/252716

Ninth Real-Time Linux Workshop 2007
http://lwn.net/Articles/260118
http://linuxdevices.com/articles/AT4991083271.html